Row-Store / Column-Store / Hybrid-Store

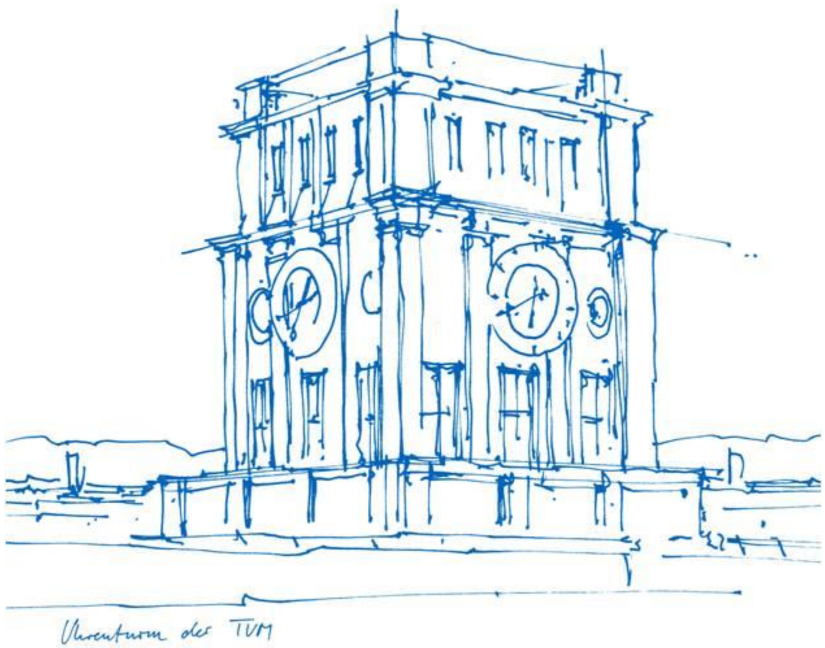
Kevin Sterjo

Technische Universität München

Department of Informatics

Chair for Database Systems

Garching, 12. December 2017



Chair for Database Systems
Department of Informatics
Technische Universität München

Table of contents

Table of contents

Introduction

Disk Resident Database Systems (DRDB)

- Data is stored on disk
- May be cached into memory for access

Main Memory Database Systems (MMDB)

- Data is stored permanently on main physical memory
- Backup on disk (if needed)

Introduction

Why use MMDB?

- Lower I/O cost
- Access time
- Directly accessible by the processor(s)
- Getting cheaper

Table of contents

Chair for Database Systems
Department of Informatics
Technische Universität München

Row Store

Traditional way data is physically stored

All attributes of each tuple stored subsequently

1st row to memory

2nd row to memory

..

R

A	B	C	D	E
01	11	21	31	41
02	12	22	32	42
03	13	23	33	43
04	14	24	34	44
05	15	25	35	45
06	16	26	36	46
07	17	27	37	47
08	18	28	38	48
09	19	29	39	49

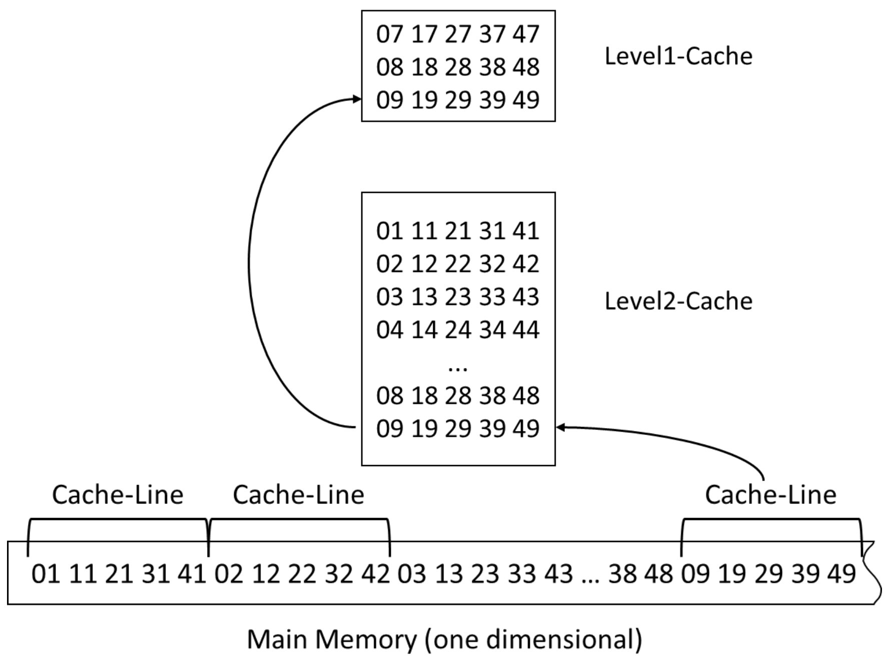


Figure 1: Logical Row Store. Source: own representation based on [1]

Row Store

Whole row written in a single operation

More preferable for OLTP-oriented databases

What happens when we need to access only one attribute?

R

A	B	C	D	E
01	11	21	31	41
02	12	22	32	42
03	13	23	33	43
04	14	24	34	44
05	15	25	35	45
06	16	26	36	46
07	17	27	37	47
08	18	28	38	48
09	19	29	39	49

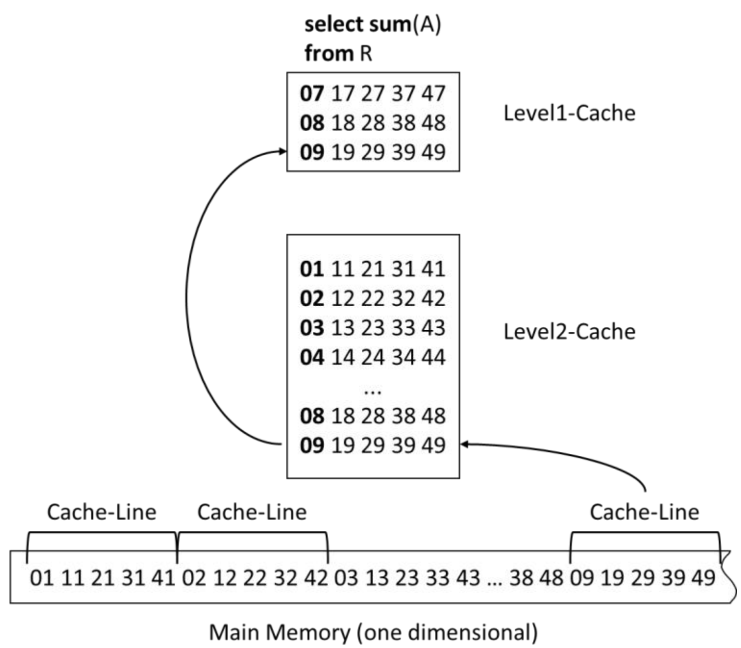


Figure 2: Logical Row Store with query. Source: own representation based on [1]

Row Store

Attributes of different types

Compression algorithms difficult to implement compared to other layouts

Use of dictionaries

Huffman encoding

Table of contents

Chair for Database Systems
Department of Informatics
Technische Universität München

Column Store

Dates back to the ’70s

Attributes depicted by columns stored contiguously

R

A	B	C	D	E	F
01	11	21	31	41	51
02	12	22	32	42	52
03	13	23	33	43	53
04	14	24	34	44	54
05	15	25	35	45	55
06	16	26	36	46	56
07	17	27	37	47	57
08	18	28	38	48	58
09	19	29	39	49	59

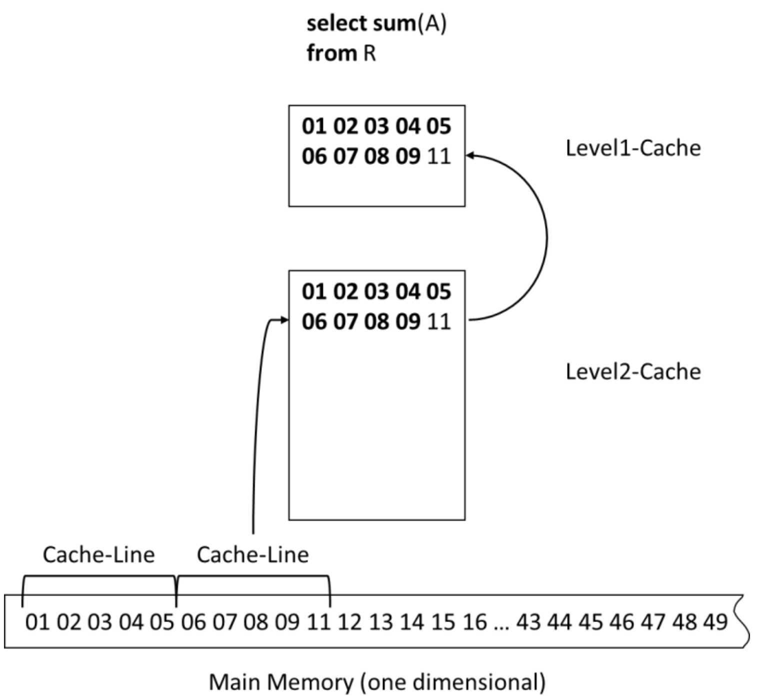


Figure 3: Logical Column Store. Source: own representation based on [1]

Chair for Database Systems
Department of Informatics
Technische Universität München

Column Store

Attributes of the same type

Perform an order of magnitude better on analytical workloads

Successfully implemented in OLAP-oriented databases

Compression also a factor

But…

Write operations and tuple construction problematic

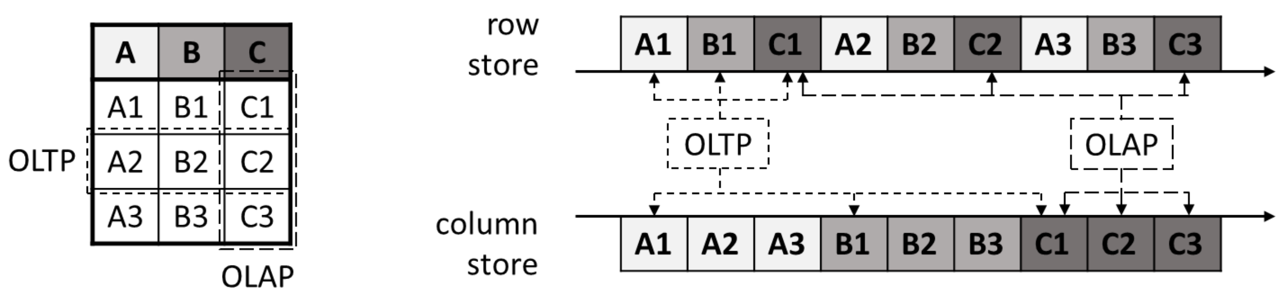


Figure 4: Memory alignment and resulting access patterns for row and column store.
Source: own representation based on [2]

Chair for Database Systems
Department of Informatics
Technische Universität München

Table of contents

Chair for Database Systems
Department of Informatics
Technische Universität München

Hybrid Store

Indecisiveness between row and column store

What happens when advantages of them are required?

Combination of both techniques

Insert and update intense data stored in row store component

Data used for analytical processes stored in column store

Hybrid Store

Transactional processing carried out on a dedicated OLTP database system

Additional Data Warehouse implemented for business intelligence query processing

ETL (Extract-Transform-Load) in specific intervals

Data staleness

Table of contents

Chair for Database Systems
Department of Informatics
Technische Universität München

Table of contents

Implementation
Initial Effort

Use of basic C++ objects, namely **“vector”** and **“struct”**
Table used provided as CSV file consisting of three columns with generated data containing stock purchase transactions: *Name (string), Quantity (int)* and *Price (float)*

```
Cruz,700,22.00
Rina,77,174.04
Caryn,62,667.50
Hop,971,342.37
Donovan,406,684.31
Caleb,602,310.12
Andrew,789,417.30
Blossom,605,992.47
Mitchel,506,31.03
Sharon,647,58.99
```

Figure 5: CSV file dataset snippet
Source: own representation

Implementation
Initial Effort

Three classes:

1. **RowStore**:
   – Row: **struct Row [Name, Quantity, Price]** object
   – Table: **vector<Row>** object containing such rows
2. **ColumnStore**:
   – Column: **vector<>** object depending on the column type
   – Table: **struct Table [vector(Name), vector(Quantity), vector(Price)]**
3. **HybridStore**:
   – Partial row: **struct MiniRow [Quantity, Price]** object
   – Table: **vector(Name)** object for the first column, and a **vector<MiniRow>** for the second "column" containing (Quantity, Price) per entry

Implementation
Initial Effort

Functions for each class:

- Insertion and selection
- **SUM(Quantity)**
- **AVG(Price)**
- **SUM(Quantity * Price)**

Table of contents

Chair for Database Systems
Department of Informatics
Technische Universität München

Implementation Evaluation

Theoretically accessing the data in three different ways
Once CSV file loaded, objects are stored in-memory to be used during runtime
Longer time for initial loading, but much shorter time to execute the aggregations
Testing with a dataset of 1000 rows almost unmeasurable

# Entries	1 Million			10 Million		
Store Type	Row Store	Column Store	Hybrid Store	Row Store	Column Store	Hybrid Store
INSERT	91929	101312	106577	1467790	1437250	1427344
SELECT	3117	3444	4084	9261	10598	7079
SUM(Quantity)	212	718	204	612	528	626
AVG(Price)	204	215	205	1250	1032	1244
SUM(Quantity*Price)	223	226	227	2386	1983	1133

Figure 6: Execution times for 1 million and 10 million entries (in milliseconds)
Source: own representation

Chair for Database Systems
Department of Informatics
Technische Universität München

Table of contents

Chair for Database Systems
Department of Informatics
Technische Universität München

Implementation Approach

New approach by using already stored data

One CSV file for each store mode: row, column and hybrid

Data already prepared as would be expected for each store

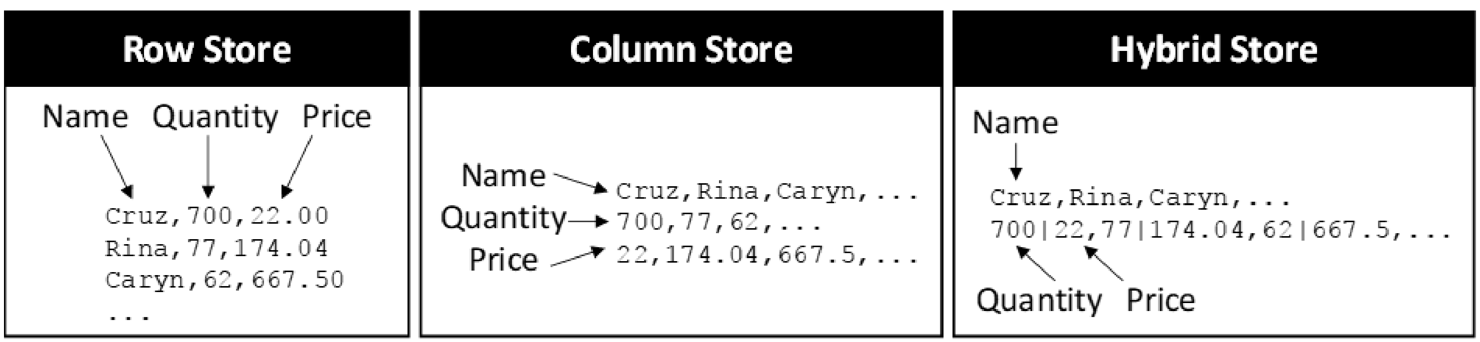


Figure 7: Input CSV file structure for the second approach
Source: own representation

Implementation Approach

Two advantages using these CSV files as “storage”:

1. Slowed down access when performing aggregations because the content is not loaded in-memory but continuously from a slower storage
2. Minimized influence of the environment factors and further internal optimization when dealing with known objects

Operation:

No **INSERT** operation in this approach

SELECT(*) (with a filter on Name)

COUNT(*)

SUM(Quantity)

AVG(Price)

SUM(Quantity*Price)

Table of contents

Chair for Database Systems
Department of Informatics
Technische Universität München

Implementation Results

Longer time for aggregation operations

# Entries	1 Million		
Store Type	Row Store	Column Store	Hybrid Store
SELECT * [Name='Zoe']	922	271	253
COUNT(*)	826	102	102
SUM(Quantity)	1482	716	1384
AVG(Price)	4673	4054	4619
SUM(Quantity*Price)	8697	7917	8426

Figure 8: Second approach execution times for 1 million entries (in milliseconds)
Source: own representation

COUNT(*) – faster in column and hybrid store
SUM(Quantity) and **AVG(Price)** – faster in column store
SUM(Quantity*Price) – slightly faster in column store
SELECT – faster in column and hybrid store

Table of contents

Questions?

References

[1] Alfons Kemper and André Eickler. *Datenbanksysteme. Eine Einführung.* 10. Auflage. De Gruyter. pp. 583-600. 2015.

[2] Philipp Rösch, Lars Dannecker, Gregor Hackenbroich and Franz Färber. *A Storage Advisor for Hybrid-Store Databases.* pp. 1748-1758. 2012.